

Funkcionalna verifikacija hardvera

Vežba 4

Randomizacija i generisanje ograničenja u SystemVerlog jeziku

Sadržaj

1	Randomizacija	4
1.1	Ugrađene funkcije	4
1.1.1	<i>\$urandom</i>	4
1.1.2	<i>\$urandom_range</i>	4
1.2	Randomizacija i OOP	4
1.2.1	Pre i post randomize metode	5
1.3	Std::randomize	6
2	Ograničenja (engl. <i>constraints</i>)	8
2.1	Pripadnost	8
2.2	Težinska distribucija	8
2.3	Uslovna ograničenja	9
2.4	Iterativna ograničenja	9
2.5	<i>In-line</i> ograničenja	10
2.6	Verovatnoće rešenja	10
2.7	Kontrola	11
2.7.1	Uključivanje i isključivanje ograničenja	11
2.7.2	Uključivanje i isključivanje randomizacije	12
2.8	Česte greške	12
3	Simulacija	14
4	Zadaci	16

Četvrta vežba je posvećena randomizaciji i ograničenjima. Preći će se osnovne funkcije za randomizaciju u SystemVerilog-u, pokazati randomizacija u klasama i osnovne konstrukcije za ograničenja. Takođe, će se pokazati kontrola random generatora u simulatoru.

Constraint-driven generisanje testova tj. generisanje testova koristeći randomizaciju sa određenim ograničenjima umnogome olakšava automatizaciju i pisanje testova za funkcionalnu verifikaciju. Direktno testiranje je dosta neefikasno. Sa porastom kompleksnosti dizajna, postaje gotovo nemoguće pokriti sve funkcionalnosti direktnim testovima. Ne samo zbog velikog broja direktnih testova koje bi trebalo napisati, već i zbog nemogućnosti sagledanja svih osobina dizajna i interakcije između njih. Randomizacija omogućava lak način kreiranja testova, pokrivanje velikog broja situacija i mnogo detaljniju verifikaciju, dok pisanje ograničenja omogućava restrikciju randomizacije na interesantne i validne scenarije. Iako je pisanje direktnih testova u početku lakše i brže, utrošak vremena za pisanje randomizovanog testa je opravdan jer se, na kraju, obavlja mnogo kvalitetnija verifikacija DUT-a.

1 Randomizacija

Postoji više načina za dobijanje nasumičnih vrednosti u SystemVerilog-u. Za dobijanje pojedinačnih vrednosti postoji veliki broj ugrađenih funkcija, međutim najveće prednosti se dobijaju koristeći random generator uz OOP.

1.1 Ugrađene funkcije

Najčešće korišćene funkcije za dobijanje nasumičnih vrednosti su `$urandom` i `$urandom_range`. Sledi njihov opis.

1.1.1 `$urandom`

`$urandom` funkcija se koristi za generisanje 32-bitnog nasumičnog broja. Dobijeni broj je *unsigned*, a prototip funkcije je:

```
function int unsigned $urandom(int seed);
```

seed predstavlja opcioni argument i određuje sekvencu random brojeva koji će biti generisani pri svakom pozivu funkcije. Pri korišćenju istog *seed*-a, random generator će uvek generisati istu sekvencu nasumičnih brojeva. `$urandom` funkcija je dosta slična `$random` funkciji, ali vraća *unsigned* brojeve i ona je stabilna za korišćenje u nitima (*threads*). Zbog toga se preporučuje korišćenje ove funkcije.

1.1.2 `$urandom_range`

`$urandom_range` funkcija vraća *unsigned* integer, ali u specificiranom opsegu. Prototip je:

```
function int unsigned $urandom_range(int unsigned maxval, int unsigned minval = 0);
```

Vraćeni broj će pripadati opsegu `[maxval : minval]`, pri čemu je *minval* argument opcion, a podrazumevana vrednost je 0. Npr:

```
x = $urandom_range(10, 4); // x dobija vrednost izmedju 4 i 10
x = $urandom_range(10); // x dobija vrednost izmedju 0 i 10
x = $urandom_range(4, 10); // x dobija vrednost izmedju 4 i 10, ukoliko je maxval < minval, argumenti se
    zamenjuju
```

1.2 Randomizacija i OOP

U svakoj klasi postoji metoda *randomize* koja služi za dobijanje nasumičnih vrednosti polja. Korišćenje klasa pruža snažan mehanizam za modelovanje podataka omogućavajući kreiranje generičkih objekata za dobijanje nasumičnih vrednosti koji se mogu lako ponovno koristiti, nasleđivati, uključivati/isključivati i sl.

Random polja u klasi se deklariraju upotrebom ključnih reči *rand* i *randc*. Ova osobina se može primeniti na bilo koji celobrojni tip, ali i na nizove i redove. Prilikom randomizacije nizova i redova randomizovaće se svaki element zasebno, pri čemu je moguće ograničiti i veličinu niza (pogledati poglavlje 2.4). Razlika između *rand* i *randc* modifikatora je sledeća:

- *rand* - standardna random varijabla. Vrednosti koje prima su uniformno raspoređene preko čitavog opsega.
- *randc* - *random-cyclic* varijabla koja iterira kroz sve vrednosti u datom opsegu bez ponavljanja. Maksimalna veličina promenljivih koje mogu biti *randc* je 16 bita. Prilikom randomizacije se pronalazi permutacija svih mogućih vrednosti, a zatim se prilikom svakog poziva vraća sledeći broj u permutaciji. Nakon prolaska kroz sve elemente, pronalazi se sledeća nasumična permutacija.

Da bi se polja randomizovala potrebno je pozvati *randomize()* metodu nad objektom. Ako se pozove bez argumenata dodeljuje nasumičnu vrednost svim poljima koja su deklarirana kao *rand* ili *randc*. Metoda vraća 1 ukoliko je randomizacija uspešna, a 0 ukoliko nije. Korisno je uvek proveriti povratnu vrednost pošto je lako moguće da randomizacija ne uspe ukoliko su ograničenja pogrešno napisana i/ili postoji više ograničenja u konfliktu (ograničenja su opisana u narednom poglavlju). Moguće je randomizovati i samo neka polja u objektu. Ta polja se prosleđuju *randomize* metodi kao argumenti. Sledi primer randomizacije *transaction* klase.

```
class transaction;

    rand bit [1 : 0] addr;
    rand bit [7 : 0] data;

    function void display_transaction();
        $display("\taddr = %0h", this.addr);
        $display("\tdata = %0h", this.data);
    endfunction : display_transaction
endclass : transaction

module top;
    transaction tr;

    initial begin
        tr = new;
        $display(" Initial ");
        tr.display_transaction();
        assert (tr.randomize());
        $display(" Randomize all");
        tr.display_transaction();
        assert (tr.randomize(data));
        $display(" Randomize just data");
        tr.display_transaction();
    end

    // Rezultat izvršavanja:
    //
    // Initial
    //   addr = 0
    //   data = 0
    // Randomize all
    //   addr = 2
    //   data = 1d
    // Randomize just data
    //   addr = 2
    //   data = 9a

endmodule : top
```

Kod 1: Primer *randomize* metode

Poziv *tr.randomize()* će dodeliti nasumičnu vrednost poljima *addr* i *data*, dok će poziv *tr.randomize(data)* randomizovati samo *data* polje, dok *addr* polje ostaje nepromenjeno.

Provera uspešnosti randomizacije se, kao i u prethodnom primeru, često vrši naredbom *assert*. Slično kao u VHDL-u, naredba *assert* proverava izraz u zagradi i ukoliko je vrednost netačna javlja grešku.

1.2.1 Pre i post randomize metode

Pored *randomize* metode, svaka klasa sadrži *pre_randomize* i *post_randomize* metode, koje se automatski pozivaju pre i posle poziva *randomize* metode, respektivno. U svakoj klasi je moguće *override*-ovati ove metode kako bi se izvršile potrebne kalkulacije, ispis vrednosti i sl.

```

class transaction;

    rand bit [1 : 0] addr;
    rand bit [7 : 0] data;

    function void display_transaction();
        $display("\taddr = %0h", this.addr);
        $display("\tdata = %0h", this.data);
    endfunction : display_transaction

    function void pre_randomize();
        $display("transaction pre_randomize:");
        this.display_transaction();
    endfunction : pre_randomize

    function void post_randomize();
        $display("transaction post_randomize:");
        this.display_transaction();
    endfunction : post_randomize

endclass : transaction

module top;
    transaction tr;

    initial begin
        tr = new;
        assert(tr.randomize());
    end

    // Rezultat izvršavanja:
    //
    // transaction pre_randomize:
    //   addr = 0
    //   data = 0
    // transaction post_randomize:
    //   addr = 2
    //   data = 1d

endmodule : top

```

Kod 2: Primer *pre_randomize* i *post_randomize* metode

Ispis posle izvršavanja datog primera je takođe dat. Primetiti da nije potrebno eksplicitno pozvati *pre/post randomize* metode, već se njihov poziv vrši automatski prilikom poziva *randomize* metode.

1.3 Std::randomize

Iako korišćenje polja u klasi pruža velike prednosti prilikom randomizacije, pojedini problemi ne zahtevaju veliku fleksibilnost i prednosti koje OOP pruža. Kada je potrebno randomizovati podatak koji ne pripada klasi moguće je koristiti *scope* funkciju za randomizaciju kako bi se randomizovala promenljiva u datom opsegu bez potrebe za definisanjem klase ili instanciranjem objekta klase. Ova funkcija je *std::randomize()*.

Način rada ove funkcije je isti kao i klasne *randomize* funkcije, s tim što ova funkcija deluje na promenljive u trenutnom *scope*-u, a ne nad članovima klase. Primer je dat u kodu 3.

```

module std_randomize;

    bit [3:0] x;
    int y;
    bit err;

    initial begin
        err = !std::randomize(x, y);
    end

```

```
    if (err)
        $display("Randomizacija nije uspesna. x = %0h, y = %0h", x, y);
    else
        $display("Randomizacija uspesna. x = %0h, y = %0h", x, y);
    end
endmodule
```

Kod 3: Primer *std::randmize* metode

Prednosti korišćenja ove funkcije, nasuprot ugrađenim *\$urandom* i *\$urandom_range*, su u mogućnosti dodavanja ograničenja (pogledati poglavlje [2.5](#)), kao i olakšanom korišćenju za kompleksnije promenljive.

2 Ograničenja (engl. *constraints*)

Puštanje testova sa čistim random vrednostima često nije zgodno. Trebalo bi previše vremena da se dođe do interesantnih scenarija, možda neke vrednosti nisu validne, možda neke nisu interesantne za proveru itd. Korišćenjem ograničenja se mogu specificirati interesantni opsezi za generisanje stimulusa.

Ograničenja se pišu u takozvanim *constraint* blokovima. *Constraint* blokovi su članovi klase, kao i polja i metode, i moraju imati jedinstveno ime, mogu se nasleđivati ili predefinisati. Blok se definiše ključnom reči *constraint* praćenom imenom ograničenja i vrednostima koje se ograničavaju u vitičastim zagradama. Npr.

```
constraint data_constraint { data > 5; }
```

Ograničenje pod imenom *data_constraint* ograničava vrednost *data* polja na vrednosti veće od 5, odnosno prilikom randomizacije *data* polja, uvek će mu se dodeliti nasumična vrednost veća od 5.

Ograničenja mogu biti i dosta kompleksnija od gore navedenog primera, i mogu sadržati veći broj naredbi i promenljivih. Sledeći primer ograničava vrednost *addr* na 1, i vrednost *data* na opseg između 5 i 10. Naredbe se odvajanju korišćenjem “;”.

```
constraint addr_data_constraint { addr == 1; data > 5; data < 10; }
```

Postoji i mnogo operatora za pisanje ograničenja koji omogućavaju lako dobijanje interesantnih vrednosti. U nastavku je dat pregled najčešće korišćenih.

2.1 Pripadnost

Koristeći *inside* operator moguće je ograničiti pripadnost promenljive na skup datih vrednosti. Sve vrednosti unutar navedenog opsega imaju jednaku verovatnoću odabira. Npr.

```
constraint data_inside_constraint { data inside { [8'h5 : 8'hA] }; }
```

Prethodni primer ograničava vrednost *data* polja na vrednosti između 5 i 10. Obratiti pažnju na sintaksu: vrednosti koje gleda *inside* operator se navode unutar vitičastih zagrada (odvojene zarezima ukoliko ih ima više) dok se intervali navode u uglastim zagradama. Sledeći primer dozvoljava i vrednosti 100 i 255 za *data* polje:

```
constraint data_inside_constraint { data inside { [8'h5 : 8'hA], 8'h64, 8'hFF }; }
```

Odnosno prilikom randomizacije *data* polje može dobiti vrednosti iz skupa {5, 6, 7, 8, 9, 10, 100, 255}, sa jednakom verovatnoćom odabira svake vrednosti.

Ovaj operator je moguće iskoristiti i za ograničavanje opsega kome vrednost ne pripada, na primer:

```
constraint data_outside_constraint { !(data inside { [8'h5 : 8'hA], 8'h64, 8'hFF }); }
```

2.2 Težinska distribucija

Ponekad je potrebno kontrolisati verovatnoću odabira pojedinih vrednosti. Ovo omogućava *dist* operator. Potrebno je operatoru proslediti listu vrednosti i težina, odvojenih sa “:=” ili “:/” operatorom. Vrednosti mogu biti pojedinačne ili opsezi, dok su težine celobrojne vrednosti. Vrednost sa većom težinom će biti češće dodeljivana nego vrednost sa manjom težinom. Zbir svih težina ne mora biti jednak 100, a podrazumevana vrednost je 1.

“:=” operator dodeljuje težinu navedenoj vrednosti, a ukoliko je u pitanju opseg, sve vrednosti unutar opsega će dobiti navedenu težinu. “:/” operator sa druge strane će podeliti težinu sa brojem elemenata u opsegu i dodeliti tu težinu svakoj vrednosti u opsegu, odnosno težina svake vrednosti u opsegu će biti $\langle \text{br_el} \rangle / \langle \text{težina} \rangle$.

```
constraint addr_dist_constraint { addr dist {2'd0 := 5, 2'd1 := 15 }; }
```

Prethodni primer ilustruje *dist* operator. Vrednosti 0 i 1 imaju težine 5 i 15, respektivno, što znači da je verovatnoća odabira vrednosti 0 za *addr* polje $5/20$, a verovatnoća odabira vrednosti 1 je $15/20$ tj. vrednost 0 će se odabrati u 25% slučajeva, a vrednost 1 u 75% slučajeva. U ovom primeru bismo isti rezultat dobili i da je korišćen “:/” operator umesto “:=” operatora, a sledeći primeri ilustruju razlike između ova dva operatora:

```
constraint addr_dist_1_constraint { addr dist {0 := 5, [1 : 3] := 15 }; }
```

Sada *addr* može poprimiti vrednosti 0, 1, 2 i 3. Težina vrednosti 0 je 5, dok svaka od vrednosti 1, 2 i 3 ima težinu 15. Ukupna težina je $5 + 3 \cdot 15 = 50$, što znači da je verovatnoća odabira nule $5/50$, jedinice $15/50$, dvojke $15/50$ i trojke $15/50$.

```
constraint addr_dist_2_constraint { addr dist {0 :/ 5, [1 : 3] :/ 15 }; }
```

Za razliku od prethodnog primera, zbog korišćenja “:/” menja se način računanja težina u intervalu. Težina vrednosti nula je i dalje 5, ali sada svaka od vrednosti 1, 2 i 3 ima težinu $15/3 = 5$. Sada je ukupna težina 20, a verovatnoća odabira svake vrednosti $5/20$.

2.3 Uslovna ograničenja

Postoji dva načina deklarisanja uslovnih ograničenja: implikacija ($\rightarrow$) i *if..else* konstrukcija. Npr.

```
constraint implication_constraint { (addr == 0) -> (data == 0); }
```

Ukoliko je vrednost *addr* polja nula, i vrednost *data* polja se ograničava na 0.

```
constraint if_else_constraint {
  if (addr == 0)
    data == 0;
  else
    data == 5;
}
```

Ukoliko je *addr* 0 i *data* je 0, a ukoliko *addr* $\neq$ 0 *data* je 5. *If..else* konstrukciju je zgodno koristiti ukoliko je potreban *else* deo, radi čitljivosti koda, iako se isto može postići i sa implikacijom.

2.4 Iterativna ograničenja

Nizove i redove je moguće ograničiti koristeći *foreach* petlju i *size* metodu. Ovo se postiže na sledeći način:

```
rand bit [7 : 0] data[$]; // queue holding 8-bit data

constraint queue_constraint {
  data.size == 5;
  foreach (data[i]) data[i] < 100;
}
```

Prethodni primer ograničava red *data* na tačno 5 elemenata pri čemu svaki element ima vrednost manju od 100.

2.5 In-line ograničenja

Prilikom svakog poziva *randomize* metode nad objektom neke klase, uzimaju se u obzir sva ograničenja koja pripadaju toj klasi. Međutim, u nekim slučajevima je potrebno dodati još neka ograničenja prilikom pojedinih poziva *randomize* metode. Ovo se postiže koristeći *in-line* ograničenja. Poziv *randomize* metode je praćen ključnom reči *with* i željenim ograničenjima. Npr.:

```
assert (tr.randomize() with { addr != 0; data > 2; } );
```

Primetiti da se, kao i za sva ograničenja, koriste vitičaste zagrade. Takođe, pošto se ograničenja nalaze u opsegu klasa, dovoljno je navesti samo imena polja *addr* i *data*, a ne *tr.addr* i *tr.data*.

Na isti način je moguće ograničiti *std::randomize* funkciju:

```
assert (std::randomize(x, y) with { x > y; } );
```

2.6 Verovatnoće rešenja

Tokom pisanja ograničenja važno je voditi računa o verovatnoći odabira nekog rešenja. Prilikom razrešavanja ograničenja mora se omogućiti uniformna distribucija nad svim kombinacijama legalnih vrednosti promenljivih, odnosno da sve kombinacije imaju jednaku verovatnoću odabira. Na primer u klasi *bez_ograničenja* datoj u kodu 4 ne uvode se ograničenja poljima *ctrl* i *data*. Postoji 2^{33} mogućih rešenja i svako ima verovatnoću $1/2^{33}$ (tabela 1).

```
class bez_ograničenja;
    rand bit ctrl;
    rand bit [31:0] data;
endclass

class implikacija;
    rand bit ctrl;
    rand bit [31:0] data;
    constraint c_impl {ctrl -> (data == 0);}
endclass

class sa_redosledom;
    rand bit ctrl;
    rand bit [31:0] data;
    constraint c_impl {ctrl -> (data == 0);}
    constraint c_red {solve ctrl before data;}
endclass
```

Kod 4: Primer *solve-before*

<i>ctrl</i>	<i>data</i>	Verovatnoća
0	'h00000000	$1/2^{33}$
0	'h00000001	$1/2^{33}$
0	'h00000002	$1/2^{33}$
...	...	...
1	'hffffffd	$1/2^{33}$
1	'hffffffe	$1/2^{33}$
1	'hffffff	$1/2^{33}$

Tabela 1: Bez ograničenja

Međutim, za neke slučajeve ovo nije željeno ponašanje. Često su neke kombinacije od većeg interesa i treba omogućiti njihovo češće pojavljivanje.

Na primer, ukoliko jednobitna promenljiva *ctrl* kontroliše 32-bitnu promenljivu *data* (kod 4, klasa *implikacija*) i želimo da *data* ima vrednost 0 kada je *ctrl* jednak 1. Ograničenje *c_impl*

nam to omogućava. Međutim ovo ograničenje ne znači da se prvo pronađe vrednost za *ctrl*, pa se na osnovu te vrednosti odabere vrednost za *data* polje. Bez nametnutog redosleda *ctrl* i *data* se određuju zajedno. Postoji ukupno $2^{32}+1$ legalnih kombinacija vrednosti (*ctrl*, *data*) koje ispunjavaju zadat uslov i sve imaju jednaku verovatnoću odabira (tabela 2).

<i>ctrl</i>	<i>data</i>	Verovatnoća
1	'h00000000	$1/(2^{32}+1)$
0	'h00000000	$1/(2^{32}+1)$
0	'h00000001	$1/(2^{32}+1)$
0	'h00000002	$1/(2^{32}+1)$
0	...	
0	'hffffffd	$1/(2^{32}+1)$
0	'hffffffe	$1/(2^{32}+1)$
0	'hffffff	$1/(2^{32}+1)$

Tabela 2: Sa ograničenjem: implikacija

Ctrl polje je jednako 1 samo za jednu od ovih kombinacija tj. kada je $(ctrl, data) = (1, 0)$ što znači da je verovatnoća da *ctrl* bude jednako 1 zapravo $1/(2^{32}+1)$ odnosno praktično nula, što verovatno nije željeno ponašanje.

U ovom slučaju želimo da se vrednost za *ctrl* bira odvojeno od *data* odnosno da se prvo razreši *ctrl*, pa tek onda *data*. *Solve-before* mehanizam nam ovo omogućava. U klasi *sa_redosledom* u kodu 4, dodali smo ograničenje *c_red* u kome eksplicitno navodimo da je potrebno razrešiti polje *ctrl* pre polja *data*. Navođenjem ovog ograničenja drastično menjamo verovatnoće odabira. Sada je verovatnoća da će *ctrl* polje imati vrednost 1 50%. Pošto *data* zavisi od *ctrl*, vrednost 0 će takođe biti odabrana u blizu 50% slučajeva. Treba napomenuti da *solve-before* mehanizam može da promeni verovatnoću odabira vrednosti, ali ne utiče na prostor rešenja tj. na legalne kombinacije promenljivih. Verovatnoće za ovaj primer su prikazane u tabeli 3.

<i>ctrl</i>	<i>data</i>	Verovatnoća
1	'h00000000	$1/2$
0	'h00000000	$1/2 * 1/2^{32}$
0	'h00000001	$1/2 * 1/2^{32}$
0	'h00000002	$1/2 * 1/2^{32}$
0	...	
0	'hffffffd	$1/2 * 1/2^{32}$
0	'hffffffe	$1/2 * 1/2^{32}$
0	'hffffff	$1/2 * 1/2^{32}$

Tabela 3: Sa ograničenjem: implikacija i *solve-before*

2.7 Kontrola

rand_mode() i *constraint_mode()* metode omogućavaju podešavanja aktivnosti promenljivih i ograničenja odnosno da li je nasumična polje aktivno ili ne i da li je ograničenje aktivno ili ne. U nastavku je dat njihov opis.

2.7.1 Uključivanje i isključivanje ograničenja

Iako se ograničenja pišu kako bi se randomizacija usmerila na scenarije od interesa, ponekad ih je potrebno isključiti. Npr. ukoliko namerno želimo da generišemo pogrešnu transakciju kako bi videli reakciju DUT-a. Ovo se postiže korišćenjem metode *constraint_mode*. Metodi se kao argument prosleđuje 0 za isključivanje ograničenja, odnosno 1 za uključivanje. Podrazumevana vrednost je da

su sva ograničenja uključena. Takođe, moguće je isključiti/uključiti sva ograničenja ili samo neka, kao što ilustruje sledeći primer:

```
class transaction;

    rand bit [1 : 0] addr;
    rand bit [7 : 0] data;

    constraint data_range { data > 'ha5; }
    constraint addr_range { addr == 0; }

endclass : transaction

module top;

    transaction tr;

    initial begin
        tr = new;
        assert (tr.randomize()); // i data_range i addr_range ograničenja su aktivna
        $display("addr = %0h, data = %0h", tr.addr, tr.data);

        tr.constraint_mode(0); // isključivanje svih ograničenja
        assert (tr.randomize()); // nema aktivnih ograničenja
        $display("addr = %0h, data = %0h", tr.addr, tr.data);

        tr.data_range.constraint_mode(1); // uključivanje jednog ograničenja
        assert (tr.randomize()); // data_range je aktivno, dok addr_range nije
        $display("addr = %0h, data = %0h", tr.addr, tr.data);
    end

    // Rezultat izvršavanja:
    //
    // addr = 0, data = c7
    // addr = 3, data = 19
    // addr = 2, data = f0

endmodule : top
```

Kod 5: Primer kontrole ograničenja

2.7.2 Uključivanje i isključivanje randomizacije

Slučno kao i ograničenja i randomizaciju nad varijablama je moguće uključiti ili isključiti korišćenjem metode *rand_mode*. Upotreba je ista kao i za *constraint_mode*.

2.8 Česte greške

U ovom poglavlju je dat pregled čestih grešaka koje se prave prilikom pisanja ograničenja, kao i preporuke za lakše korišćenje.

- Koristiti samo jedan odnosni operator (<, >, ≥, ≤, ==) unutar izraza u ograničenju. Korišćenje više može dovesti do neočekivanih rezultata. Npr. naredna dva primera će rezultovati različitim rešenjima:

```
constraint bad_example { 5 < a < b; }
constraint good_example { 5 < a; a < b; }
```

Izraz u *constraint* bloku se se rešava kao svi izrazi u SystemVerilog-u, odnosno operatori će se proveravati s leva na desno, pri čemu će rezultat biti jedan bit.

- Neočekivane vrednosti se mogu javiti zbog korišćenja *signed* promenljivih ili ukoliko se desi *wrap-around*.

```
rand byte a, b;
constraint signed_example { a + b == 64; }
```

U ovom primeru a i b mogu dobiti vrednosti npr. (100, -36), što možda nije očekivano.

- Uvek proveriti rezultat randomizacije. Zbog velikog broja ograničenja u verifikacionom okruženju i mogućnosti dodavanja novih ograničenja tokom rada, moguće je da randomizacija neće uspeti.

```
if (!tr.randomize()) $error("Randomization of tr failed");
assert(tr.randomize());
```

Takođe je moguće, posle ručne izmene vrednosti nekog polja, proveriti da li objekat i dalje zadovoljava sva ograničenja. Ovo se postiže prosleđivanjem *null* vrednosti *randomize* metodi. Tada se sva polja tretiraju kao *nonrandom* i samo se proverava zadovoljivost ograničenja.

```
assert(tr.randomize(null));
```

- Korišćenje istih imena u različitim *scope*-ovima može dovesti do problema. U primeru ispod želimo da *addr* polje u objektu *tr* dobije vrednost *addr* iz modula. Da bi smo to postigli potrebno je koristiti "local:" (*scope resolution operator*). Drugi način je izbeći ponavljanje naziva, odnosno ne nazivati promenljive istim imenom ukoliko će se koristiti za međusobno ograničavanje.

```
module top;

    transaction tr;
    bit [1 : 0] addr;
    bit [1 : 0] new_addr;

    initial begin
        tr = new;
        addr = 0;
        new_addr = 0;

        assert(tr.randomize() with { addr == addr; } ); // greska!
        assert(tr.randomize() with { addr == local::addr; } );
        assert(tr.randomize() with { addr == new_addr; } );
    end

endmodule : top
```

Kod 6: Primer ceste greske

3 Simulacija

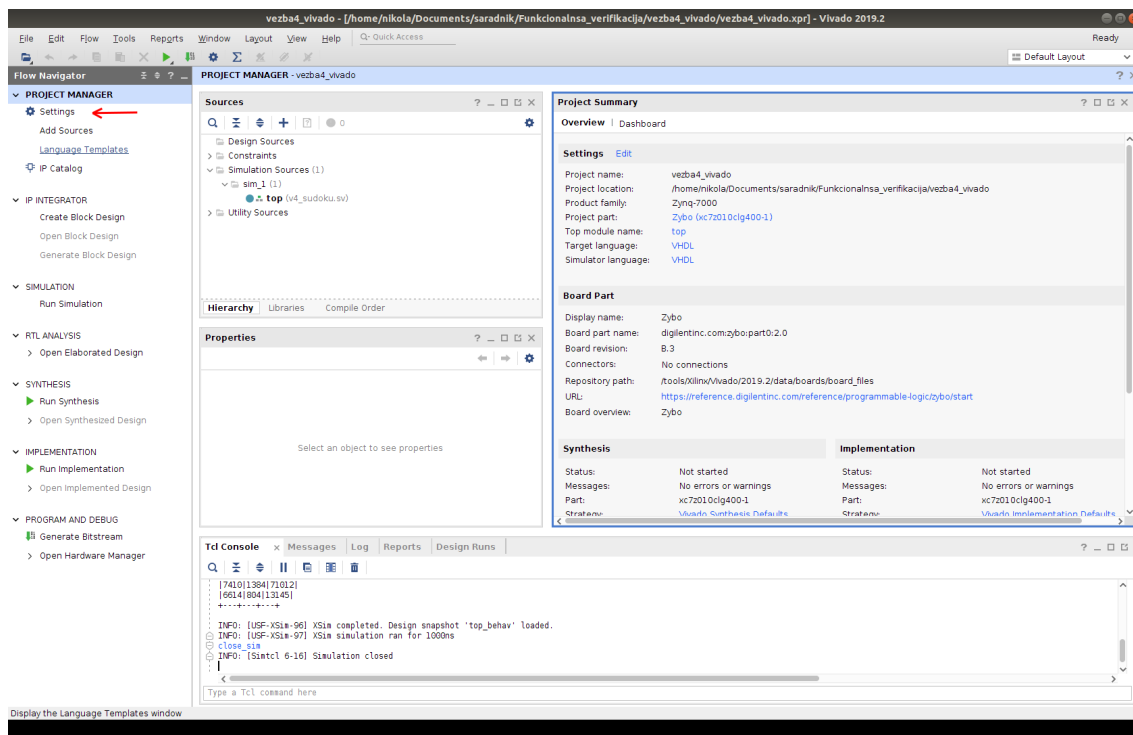
Kao što je već navedeno, *random seed* je broj koji služi za inicijalizaciju pseudo-random generatora. Pri korišćenju istog *seed-a*, pseudo-random generator će uvek generisati istu sekvencu nasumičnih brojeva. Prilikom verifikacije, uglavnom želimo da puštamo istu grupu testova sa različitim *seed*-ovima kako bismo imali veću pokrivenost. Međutim, puštanje testova sa istim *seed*-om je takođe ponekad korisno i neophodno. Npr. ukoliko je pronađena greška koja se javlja jedino pri određenom *seed*-u (odnosno jedino pri određenoj kombinaciji nasumičnih vrednosti u testu), potrebno je da se ona repodukuje, ali i proveri nakon ispravke.

Podešavanje *seed-a* u Xcelium simulatoru se vrši prilikom startovanja simulacije, prosleđivanjem *sv_seed* argumenta praćenom vrednosti *seed-a*. Ta vrednost može biti bilo koji integer ili reč *random* ukoliko želimo da se *seed* nasumično odabere. Npr:

```
xrun top_module.sv -svseed 5
xrun top_module.sv -svseed random
```

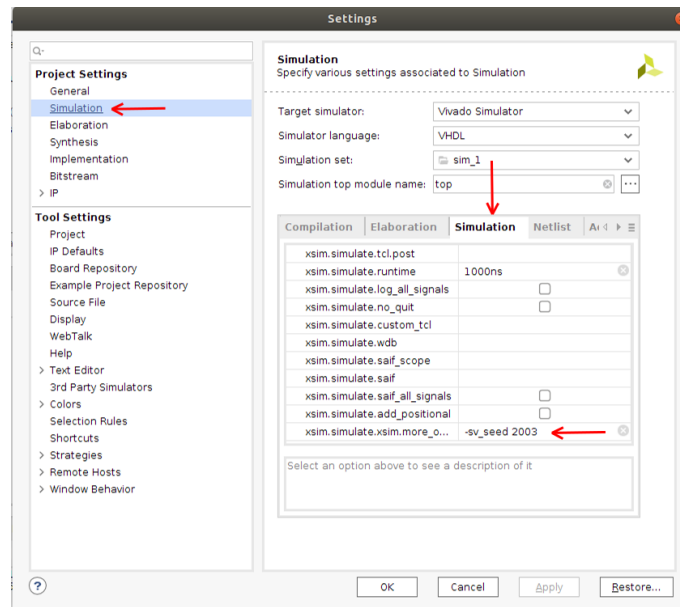
Prilikom korišćenja *random seed-a*, Xcelium simulator će ispisati koja je vrednost odabrana, kako bismo mogli da sačuvamo tu informaciju ukoliko nam bude potrebna.

Ukoliko se koristi simulator Vivado alata, neophodno je uraditi sledeće:



Slika 1: vivado settings

Odabirom opcije settings kao što je prikazano na slici 1 otvara se prozor prikazan na slici 2



Slika 2: vivado simulation settings

Odabrati opcije prikazane crvenim strelicama, i u polje *xsim.simulate.xsim.more_options* upisati:

```
-sv_seed <broj_seed-a>
```

Napomena: Ukoliko je simulacija prethodno pokrenuta, i ponovo je izvršena promena *seed-a*, neophodno je zatvoriti simulacioni prozor i ponovo pokrenuti simulaciju.

4 Zadaci

Zadatak Za primer *transaction* klase, ograničiti vrednost *addr* polja na 2 ukoliko se *data* polje nalazi u intervalu 20-50.

Zadatak Za primer *transaction* klase, dodati polje *read_write* koje će imati vrednost *read* u 75% slučajeva.

Zadatak Za primer *transaction* klase, ograničiti *data* polje tako da se ne nalazi u intervalu 20-50.

Zadatak Za primer *transaction* klase, dodati kontrolno polje (bit *parity*) koje će predstavljati parnost *data* polja i ograničiti ga na ispravne vrednosti (odabrati *even* ili *odd parity*).

Zadatak Za primer *transaction* klase, ukoliko je u pitanju *read* transakcija ograničiti adresu na vrednosti 0 ili 3, a *data* neka dobija vrednost 0 u 12% slučajeva, a neka je u intervalu 100-200 u 88% slučajeva. Ukoliko je u pitanju *write* transakcija onda je vrednost podatka uvek veća od adrese i manja od 10.

Zadatak Koristeći randomizaciju i ograničenja implementirati klasu koja služi za rešavanje sudoku igre. Zadatak je u uneti brojeve od 1 do 9 u 9x9 mrežu tako da se brojevi ne ponavljaju u redu, koloni ili 3x3 kvadratu. Nekoliko brojeva je zadato unapred. Randomizovati preostala polja i napisati ograničenja tako da se igra ispravno reši. Kod 7 daje kostur za potrebnu klasu.

```
class Sudoku;

    bit unsigned [3:0] init [9][9];
    rand bit [3:0] box [9][9];

    // vrednost brojeva je izmedju 1 i 9
    constraint box_c { /* TODO */ }

    // kvadrati u jednom redu moraju imati jedinstvene vrednosti
    constraint row_c { /* TODO */ }

    // kvadrati u jednoj koloni moraju imati jedinstvene vrednosti
    constraint column_c { /* TODO */ }

    // unutar svakog kvadrata moraju biti jedinstvene vrednosti
    constraint block_c { /* TODO */ }

    // ukoliko je zadata pocetna konfiguracija, ona mora biti ispostovana
    // broj je zadat ukoliko je init [red][kolona] != 0
    constraint init_c { /* TODO */ }

    function int solve_puzzle(bit [3:0] init [9][9]) ;
        this.init = init;
        return this.randomize();
    endfunction: solve_puzzle

    function string sprint ();
        string s = { {3{"+"}, {3{"-"}} }, "+\n"};
        for(int i = 0; i < 9; i++) begin
            s = {s, "|"};
            for (int j = 0; j < 9; j++) begin
                s = {s, $sprintf("%1d", box[i][j])};
                if (j % 3 == 2)
                    s = {s, "|"};
                if (j == 8)
                    s = {s, "\n"};
            end
            if (i % 3 == 2)
                s = {s, {3{"+"}, {3{"-"}} }, "+\n"};
        end
    end
```

```

        end
        return s;
    endfunction : sprint
endclass: Sudoku

module top;

    bit [3:0] init [9][9];
    Sudoku s;

    initial begin
        // neupisana polja = 0
        init = '{ { 0,2,3, 4,0,6, 7,8,0 },
                  { 4,0,6, 7,0,9, 1,0,3 },
                  { 7,8,0, 0,2,0, 0,5,6 },

                  { 2,3,0, 0,6,0, 0,9,1 },
                  { 0,0,7, 8,0,1, 2,0,0 },
                  { 8,9,0, 0,3,0, 0,6,7 },

                  { 3,4,0, 0,7,0, 0,1,2 },
                  { 6,0,8, 9,0,2, 3,0,5 },
                  { 0,1,2, 3,0,5, 6,7,0 } };

        s = new;
        if (s.solve_puzzle(init)) begin
            $display("Resenje je\n%s", s.sprint );
        end else begin
            $display("Nije moguće resiti problem");
        end
    end
endmodule: top

```

Kod 7: Sudoku